Image Comparison Report - TensorFlow & PyTorch

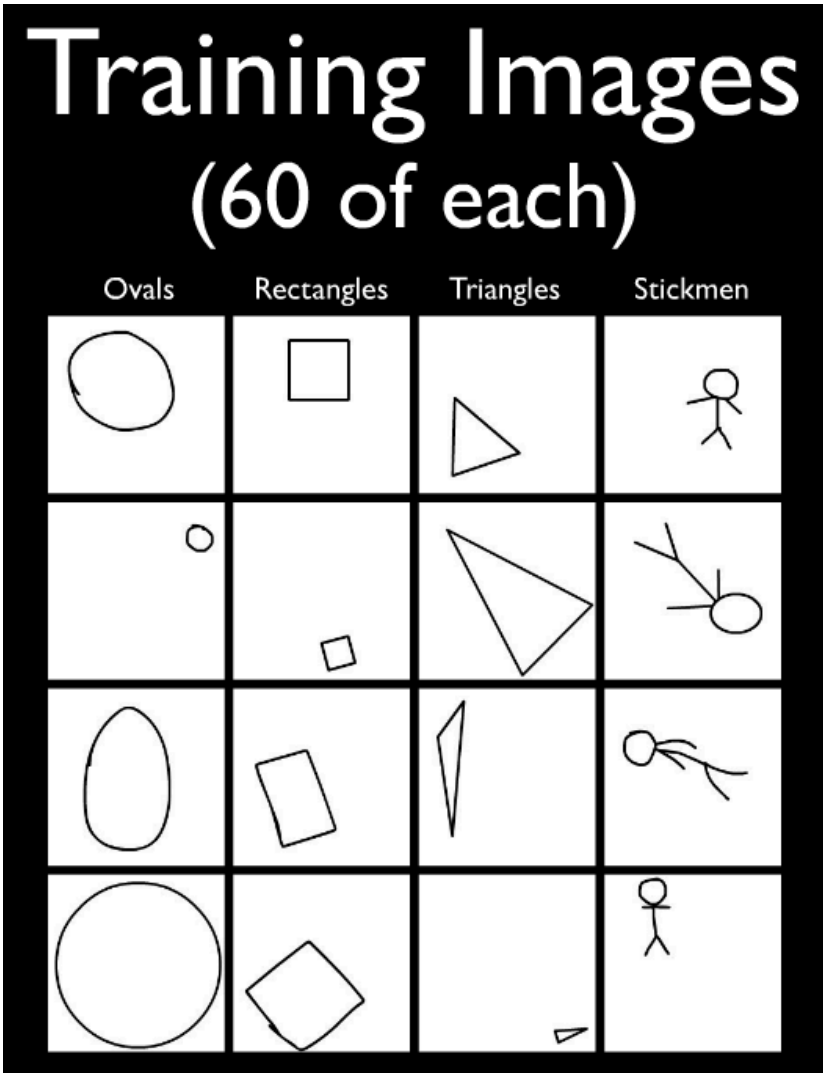
Chris Sinclair

February 25th, 2024

This report serves as a practice and comparison between two machine learning libraries using my own custom dataset: TensorFlow and PyTorch. From the outset, I want to clarify that this isn't nearly as much of a comparison of how good or accurate these libraries are. As a newcomer to machine learning, this was more of an exercise in understanding the nuances of how I would go about each step of the process in each library using this custom dataset. From there, I will compare and contrast results and then share my final thoughts on my experiences with each. But first, let's take a look at the dataset…

---

1. The Dataset



The sample above is the dataset that I put together for this project. There are four different classes (ovals, rectangles, triangles, and stick figures). The training dataset consisted of 60 images of each, while the validation set consisted of 5 images of each – 260 images total. There are a few things to note about this dataset:

1.) Some of these shapes and drawings were drawn freehand while others were made using the basic shape/line tools.
2.) The size and angle of the figure is different with each image so that the model would not bias itself with, for example, only photos of the stick figure standing right side up with its feet planted on the bottom of the image.
3.) With the goal of testing the stick figure portion, I also drew some more "natural" poses to see if the learning would "stick" even if the pose was slightly different than the norm. The main example would be the stick figure in the third row.

Each of these alterations was important in avoiding bias. These kinds of changes, such as size and angle, can also be altered during the preprocessing of the dataset but since I wanted to keep things as simple as possible, I figured it best to just do it manually.

Once all of my images were set up properly in their directories, I also set up a control dataset that I could compare results to. This was a Playing Card dataset from Kaggle with over 100 images of each card on a full deck of cards, including the Joker. Compared to the total amount of images in my custom dataset (260), the playing card dataset amounted to a total of 7624 images.

---

2. Loading & Preprocessing the Data

1.) TensorFlow

Since I was stepping into the machine learning world with no expertise with either library, I looked to the famous MNIST dataset to get an idea of how I should format my images. Notably, the TensorFlow Keras API allows a swift import of those datasets, already pre-processed and formatted so that a user can jump straight to the training model after confirming that the images and labels display properly.

However, since I was using a custom dataset, I actually found TensorFlow's image importing mechanisms to be inconvenient. In using the `image_dataset_from_directory` as well as the `flow_from_directory` in tandem with `ImageDataGenerator`, I found it very difficult to ultimately pull the images in with the same format as the MNIST dataset, which can be smoothly imported from Keras. The returns of the MNIST dataset from the Keras API's `load_data()` function are four NumPy arrays: `train_data`, `train_labels`, `test_data`, and `test_labels`. From there, this data could be easily viewed and analyzed without any further processing. On the flipside, the return of the aforementioned Keras functions are all `tf.data.Dataset` objects. Although they read properly ("`Found 240 files belonging`

`to 4 classes.`"), I was not able to conveniently view the data in any practical way that I knew of (again, I'm kind of a newbie to this stuff).

I was actually so frustrated with this that I ended up resorting to `import os` so that I could manually loop through the directory to pull the data and assign images to their proper labels. When I finally had finished it up and properly plotted the images with the same code as what I was seeing in examples, I figured I could smoothly move onward to fitting the datasets to the training model. However, it ended up either not fitting properly, or when it did fit, the amount of time spent on each epoch was only a fraction of a second – 1ms. Given that there were hundreds of images to go through, something was very wrong and I ultimately was not able to figure it out.

With this in mind, I ended up pivoting back to the `image_dataset_from_directory` method and utilized `PIL` to view the images that I had pulled. From there, I still had to normalize the data and optimize it so it would be ready to fit to a model.

2.) PyTorch

The process of loading and preprocessing my dataset in PyTorch was the complete opposite of my experience with TensorFlow. I simply used the `Dataset` class with a few small recommended changes for custom data that they noted in their documentation. From there, I just had to apply some very basic transforms and then I was ready to create a `DataLoader` out of the entire dataset. Once that was finished, my beautiful dataset was ready to be fit to the training model. Compared to TensorFlow, this was an incredibly streamlined experience that didn't even take an hour to figure out.

---

3. Building the Training Model

As mentioned in the introduction, the purpose of this project was not to compare and contrast the training models themselves. I should mention that I avoided using a pretrained model for both of them, but seeing as I'm too much of a beginner to pretend to understand how the different models and layers function, I will avoid speaking out of turn until I understand these processes more. That will have to be after a few Calculus courses and more studies into building neural networks with only the NumPy and Pandas libraries. Regardless, I will compare the overall experience with the training models in each library.

1.) TensorFlow

My experience with TensorFlow during this part of the ML process was actually, for the most part, quite pleasant. TensorFlow has abstracted each layer into the `tf.keras.layers` library, which allows a wide range of customization to the models. It's a very straightforward process where I only had to build the model (Sequential), compile it, and then fit the data to it.

2.) PyTorch

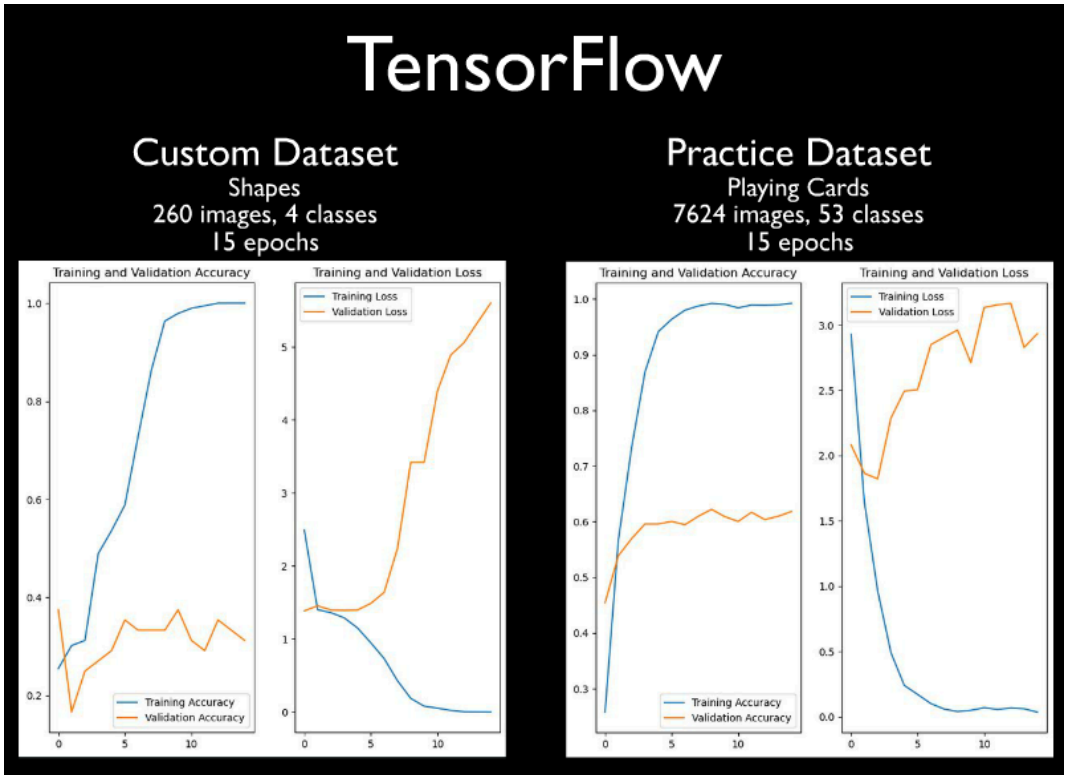
With PyTorch, I not only had to build the model, but I also needed to create the training and validation loops in order to track any specific metrics I was interested in. Although the training loop is somewhat abstracted away with functions such as `loss.backward()` and `optimizer.step()`, the control that I was able to practice in pulling metrics as well as customizing the output with every epoch was actually pretty fun to tinker with.

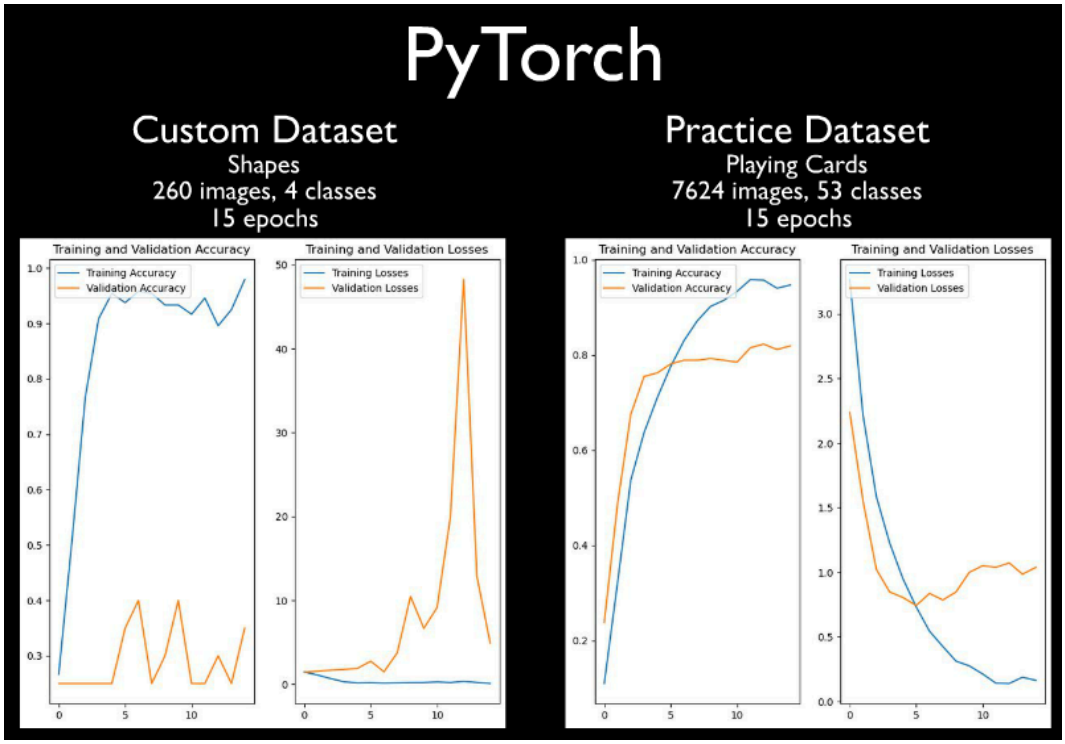
If I had to give an honest, beginner's-perspective opinion between these two, I would say that building a model in TensorFlow is a bit easier and requires significantly less code, but PyTorch gives you more control as well as understanding of the process due to being less abstracted away.

---

4. Evaluating the Results

So these results are really just a demonstration of a basic evaluation of any model. Again, I want to stress that I'm not trying to compare the quality of learning that these models are capable of. I'm more interested in sharing my experience while I essentially tried to create the same project in each library. Below are the results of my custom dataset alongside the practice dataset so that I could confirm I had done everything correctly:





As you can see, the custom dataset performed significantly worse, which I would attribute to the small sample size. With only 60 images per class, each class is a lot less information for the model to train on whereas the practice dataset had 7624 images total, which divides to about 144 images per class. That explains the low quality accuracy results for my custom dataset. However, the strong and consistent patterns for the practice dataset show that I did manage to properly train these models with both libraries.

Despite the fact that my custom dataset was not adequate to train these models on distinguishing between shapes, I still consider this project a success in learning the basics of how to approach an overall ML project in both TensorFlow and PyTorch.

Final Thoughts

After thoroughly going over each step of the basic ML process using each of these libraries, I honestly think I prefer PyTorch more. Most of this viewpoint is due to the fact that I had so much trouble loading and preprocessing my custom dataset in TensorFlow. The fact that I was ultimately unable to load and process my data into the same format as the MNIST dataset and went with an entirely different approach, in my opinion, speaks volumes about TensorFlow's ease of use for beginners. On the other hand, PyTorch had my dataset ready to go within minutes.

It could be argued that TensorFlow's model being abstracted away is also one of its benefits. However, I actually preferred the PyTorch approach of creating the training and validation loops. Sure, there was more code, but my understanding of what was happening

under the hood definitely felt like it sunk in more. Having seen it and done it firsthand now, I can appreciate why the research community has generally gravitated towards PyTorch since it's so easy to define the training/validation loops for the purpose of pulling any kind of specific metrics one would want.

In conclusion, I definitely prefer PyTorch.